Turing Machines

Reading

Section 3.1

Practice problems (page 159): 3.5, 3.8, 3.15

Turing Machines

Turing Machines (TM) are a computational structure that is quite a step up from pushdown automata in terms of capabilities. In fact in a certain sense they are strong enough to capture the idea of computation in general.

Here are some comparisons between Turing Machines and PDAs/DFAs:

- There is no separate input and stack space. There is a single infinitely long "tape", which is initially populated by the input string.
- The TM can freely move along the tape in either direction. It is not restricted by input order and stack discipline.
- Like DFAs and PDAs, the TM has a finite set of states that dictate its options.
- The TM's transition function is deterministic. We will in fact prove that adding nondeterminism does not increase the TM's computational power.
- The TM has special "accept" and "reject" states, and the computation ends the moment those are entered.
- The computation can go on forever. For CFGs in Chomsky Normal form there are bounds to the number of steps needed to reach a string of a given length. Nothing equivalent holds for TMs.

Turing Machines operate on an infinitely long tape, and at any given time the machine is pointing at some location in the tape, and is at some state. To transition, it can make a move left or right.

Here is the formal definition:

Turing Machines

A **Turing Machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where:

1. Q is a finite nonempty set of states.
2. Σ is a finite **input alphabet**, not containg a special "blank symbol".
3. Γ is a finite **tape alphabet**, containing $\Sigma \subset \Gamma$ and a special **blank symbol** $\sqcup$ representing the lack of a value at that location in the "tape".
4. $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a transition function. Based on the value at the current location in the tape, as well as the current state, the automaton can change the value at the location, switch to a new state, and also move the pointer/head to the left or to the right (L/R).

5. $q_0 \in Q$ is a start state.
6. $q_{\text{accept}} \in Q$ is the accept state
7. $q_{\text{reject}} \in Q$ is the reject state, *distinct* from q_{accept}.

Computation in a Turing Machine

- The initial input to the machine is placed at the leftmost n squares of the tape. The rest of the tape is filled with the blank symbol.
- The TM starts at the state q_0, and with the marker pointing at the leftmost location.
- The TM follows the rules dictated by the transition function δ, until it reaches the accept or reject states.
- If the TM ever tries to move to the left of the leftmost square, it just stays in that square.

We often describe Turing machines more informally than listing states and transitions, by describing the steps to be performed, with an understanding that if needed we could write down states and transitions to carry out those steps. Let us look at an example of such an informal description:

Design a TM that recognizes the language $L = \{w\#w \mid w \in \{0,1\}^*\}$.

You will recall that this was one of the languages that pushdown automata could not handle.

Here's how this TM could go:

1. We assume that our tape alphabet Γ contains "dotted" versions of our symbols, $\dot{0}, \dot{1}$. We can use those to mark locations.
2. At the beginning we are the leftmost element. We replace it with a dotted version of it, and move to a state that remembers what that value was and searches for the $\#$ marker.
3. After we reach the marker, we look at the location after it and whether it matched the value from the other side, that we were keeping track of. If it does not, then we move to reject state. If it does, we mark the location, either via a dotted version of the value or via an x.
4. Now we move left till we find the $\#$ marker, then keep moving till we find the dotted value. We move to the value after it, (i.e. the first non-dotted value), dot that and remember it, then move to find its partner on the right side.
5. We continue this back and forth until the point that we have looked at all the elements before the $\#$ marker. This means we have matched everything to the left of the $\#$ marker with the corresponding spots on the right.
6. All that is left is to travel to the right past all the x-ed out values, and see if the next thing is the blank, or if there are still more values out there. If it is blank, it means we have matched the string before $\#$ with the string after it, and so we move to accept. If not, we move to reject.

We could imagine using a dozen or so different states to carry out the above steps.

Let us try a variation:

> Design a TM that recognizes the language $L = \{ww \mid w \in \{0,1\}^*\}$.

In this instance we don't have the marker to separate the two words. We will need to find the middle of the string ourselves.

1. Start at the leftmost element, dot it, and start moving right till you find the other end.
2. Dot that element, and move left till you find that previously dotted element. dot the one after it.
3. Keep going back and forth marking the elements you have seen, one on each end.
4. When we reach the middle area, we can detect if the string had odd length: We just dotted the middle element as belonging to the left half of the string, and when we go right to find an undotted element there isn't one remaining. We can transition to a reject state right away in this case.
5. Otherwise we have determined the string has even length, and we are standing at the beginning of the right half of it. We can now proceed in two ways:
6. One way is to undot that element and remember what it is, then move all the way at the beginning of the string (should have dotted that first element in a more special way) and see if it matches, and if it does cross it out. Then take the next element, cross it out and remember it and look for it on the second half of the string, and so on.
7. Another alternative is to insert a new marker at that middle position. We can then "shift" the second half of the string one square to the right, as follows:
   - Place the marker, and remember what was there before, move to the right.
   - Place in that square the element you are remembering, and remember the new element, move to the right.
   - Keep doing that until you have arrived at the blank symbol, marking the end of the string. You then insert that last element, and transition to whatever you need to do next.
   - Now our string has obtained the form described in the previous problem, so we can use the same method we used for that one.

This idea of reusing a previously constructed TM as a helper is a key tool in the methodology of Turing Machines.

Before we move on to more examples, there is one important distinction we need to make.

> A TM on a given input can result in one of three outcomes: *accept*, *reject* or *loop*. A **loop** means that the machine goes on forever, and never reaches an accept or reject state. Unlike with DFAs and PDAs, it is not at all easy to detect the cases where the TM will loop.

We say that a TM **recognizes** a language L, if the TM results in *accept* for all strings in the language, and not in accept for all strings not in the language. So it may be the case that the TM results in a loop for some strings not in the language, and that is OK. We say that the language L is **Turing-recognizable**, if there is a TM recognizing it.

We say that a TM **decides** a language L, if the TM results in *accept* for all strings in the language and in *reject* for all strings not in the language. In other words, the TM never loops. Such a TM is called a **decider**. We call a language L **Turing-decidable**, or just **decidable**, if some Turing machine decides it.

It should be clear that every decidable language is recognizable. The interesting fact is that the converse is not true: There are languages that are recognizable, but not decidable. The examples we have seen above are examples of decidable languages.

Practice: Describe TMs that decide the following languages:

1. $A = \{a^n b^n c^n \mid n \geq 0\}$
2. $B = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$
3. $C = \{a^i b^j \mid i$ is a positive multiple of $j\}$
4. $D = \{w\#t \mid w, t \in \{a, b\}^*, w$ is a substring of $t\}$
5. $E = \{w \in \{0, 1\}^* \mid$ equal number of 0s, 1s$\}$

Turing Machine Variants

The remarkable fact about Turing Machines is that they are remarkably robust: Any variant we try turns out to have the same computational power as Turing Machines.

Stay option

A first simple variant is to allow three posible "moves", instead of just left and right. There is also a S move, for staying at the same location.

This of course isn't a considerable change: We can replace any "stay" move with a combination of a right move and a left move.

The essence of showing a variant is equivalent with the basic definition is to simulate one with the other.

Multitape machines

A more adventurous variant is one where we use multiple tapes. Each tape has its own head for reading and writing. Initially the input is all in the first tape, and the

remaining tapes are blank. The transition function would depend on the values at all heads (at all the tapes), and will update all tapes at once:

$$\delta \colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R(, S)\}^k$$

It is clear that this notion is at least as strong as that of normal TMs. We will in fact show that multitape TMs are not stronger than normal TMs.

> Every multitape Turing Machine has an equivalent single-tape Turing Machine.